Wrocław 20 maja 2015

Kamil Cała  
209954  
Środa 7:15 TN

Sprawozdanie z laboratorium nr 5

*Data laboratorium: 06.05.2015r  
Rok akademicki 2014/2015, Informatyka  
Prowadzący: Mgr. Aleksandra Postawka*

# Opis i cel ćwiczenia

Celem tego ćwiczenia było zapoznanie się z obsługą jednostki zmiennoprzecinkowej procesorów architektury x86. Złożyły się na to umiejętności odczytu i zapisu wartości do rejestru kontrolnego jednostki, oraz praktycznego zastosowania operacji zmiennoprzecinkowych w praktyce.

Ćwiczenie składało się z 2 zadań

1. Napisać w ASM funkcje pozwalające na
   1. sprawdzenie
   2. zmianę precyzji (FPU)
   3. +program w C używający funkcji w kolejności: a->b->a
2. Napisać w ASM program obliczający aproksymację funkcji sinus (szereg Taylora). Parametry: kąt i liczba kroków

# Zadanie 1 - funkcje działające na rejestrze kontrolnym FPU

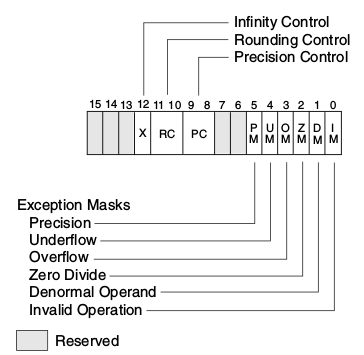
Pierwsze zadanie składa się z 3 “składników”. Są to dwa pliki zawierające funkcje w ASM oraz program w C.

## int checkPrecision()

Pierwsza funkcja, to int checkPrecision(). Funkcja ta pobiera wartość rejestru statusu FPU do pamięci, za pomocą odpowiedniej komendy:

fstcw result #load fpu status word

Aktualna precyzja wykonywanych przez FPU obliczeń zapisana jest na 8 i 9 bicie rejestru statusu.



Są to odpowiednio (za dokumentacją Intela):

|  |  |
| --- | --- |
| Pole PC | Precyzja |
| 00 | Single Precision(24 bity) |
| 01 | Reserved (wartość nieprawidłowa) |
| 10 | Double Precision (53 bity) |
| 11 | Double Extended Precision (64 bity) |

Aby zwrócić wartość tych bitów, funkcja używa maski bitowej, gdzie wszystkie z 16 bitów zostają wyzerowane, za wyjątkiem 8 i 9 bitu, które przyjmuja wartość 1. Maska wygląda wiec w zapisei binarnym nastepująco:  
0000001100000000

Po przeliczeniu na system szesnastkowy:

0x0300

Taką też więc wartość wczytujemy do rejestru przechowywującego maskę, po czym przeprowadzamy operację AND, jako parametry podając maskę oraz wczytany wcześniej rejestr stanu. Wynik należy przesunąć 8 razy w prawo aby reprezentował faktyczną wartość pola PC.

xorq %rcx, %rcx #zeroe the rcx register

movw $0x0300, %cx #use as mask for 8 and 9 bit

andw %cx, %ax #mask the status word

shr $8, %ax

ret

Funkcja zwraca ostatecznie wartość pola PC. Za wyświetlenie dla użytkownika co dane pole oznacza, odpowiedzialny jest już program główny napisany w C.

## void setPrecision(int)

Druga funkcja działa odwrotnie do pierwszej i ustawia wartość rejestru PC na taką jaka została podana w argumencie. Odczyt aktualnej wartości rejestru statusu odbywa się analogicznie jak w poprzedniej funkcji. Następnie bity pola PC są kolejno zerowane i odpowiednio ustawiane. Ponownie, przeprowadzone jest to za pomocą masek oraz instrukcji, oraz w tym wypadku - instrukcji XOR.

#zeroe the 8th and 9th bits

movw $0xfcff, %cx

andw %cx, %ax

xorq %rcx, %rcx

movw %di, %cx #use as mask for 8 and 9 bit

shl $8, %cx

xorw %cx, %ax #mask the status word

## Główny program

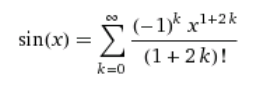
Główny program napisany w C kolejno odczytuje wartość pola PC, nadpisuje go inną wartością (która jest zapisana na twardo w kodzie, aby nie utrudniać niepotrzebnie tego pokazowego programu), oraz ponownie odczytuje wartość pola PC, aby pokazać że wartość została zmieniona. Dodatkowo program został wyposażony w funkcję  
int clarifyPrecision(int precision),

która po każdym odczycie wartości rejestru PC wyświetla w konsoli co oznacza jego wartość.

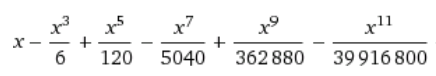
# 

# 

# Zadanie 2 - program obliczający aproksymację sinusa

Ten program w praktyczny sposób wykorzystuje FPU, do obliczenia przybliżonej wartości funkcji sinus. Całą teorię matematyczną można w tym wypadku sprowadzić do tego, że dla danego kąta x (podanego w radianach), oraz liczby kroków algorytmu k przybliżona wartość będzie wynosiła  


Co przykładowo dla k=6 da nam wzór:



Algorytm obliczania przybliżonej wartośći funkcji sinus można więc zapisać następująco w pseudokodzie:

SUMA:= x

GORA:= X

DOL:= 1

NAJ\_SILNIA:= 1

k-1 razy przeprowadź:

GORA\*=x

GORA\*=x

NAJ\_SILNIA++

DOL\*=NAJ\_SILNIA

NAJ\_SILNIA++

DOL\*=NAJ\_SILNIA

ULAMEK:= GORA/DOL

JESLI k parzyste

SUMA += ULAMEK

JESLI k nieparzyste

SUMA -= ULAMEK

KONIEC PETLI

Jak widać, aby uniknąć dużej liczby obliczeń, szczególnie dla dużych wartości k, liczba na górze ułamka za każdym razem jest przemnażana dwa razy przez x, a liczba na dole jest dwukrotnie przemnażana przez kolejne liczby silni. Dzięki temu obie liczby są stopniowo powiększane a nie obliczane od nowa za każdym razem. Wartości w szeregu Taylora są raz dodatnie a raz ujemne, więc w zależności od parzystości licznika k dodajemy, albo odejmujemy obliczony ułamek od ostatecznego wyniku.

Na początku program pobiera od użykownika parametry przy użyciu funkcji scanf. Warto zauważyć ze kat podawany jest w stopniach, a należy go przeliczyc na radiany, co można wykonać poprzez przemnożenie go przez wartość 0.0174532925 (bardzo dokładne przybliżenie). Ładujemy więc wartość kąta do rejestru zmiennoprzecinkowego. Po drodze musi ona zostać wczytana na stos, gdyż bezpośrednie wczytywanie wartości z rejestrów ogólnego użytku do zmiennoprzecinkowych jest niemożliwe. Następnie jest ona mnożona przez wspomnianą wcześniej stałą i zostaje już w stosie rejestrów zmiennoprzecinkowych do dalszego użycia.

fild -8(%rbp) #load degrees to floating point register stack

fmul DegToRad #change degrees to rads

Następnie wszystkie inne potrzebne wartości i stałe wczytywane są do rejestrów - zarówno zmiennoprzecinkowych jak i ogólnego zastosowania.

fstl 8(%rsp) #copy rads to stack

fldl 8(%rsp) #copy again to floating stack for future use as POWER

fldl 8(%rsp) #copy again to floating stack for future use as SUM

movq $1, %r10 #factorial

movq $1, %r11 #highestSoFar

movq $1, %r12 #i counter

Następnie rozpoczyna się pętla. Liczba jej iteracji kontrolowana jest przez zmienną zawartą w rejestrze %r12, która po każdym (i przed pierwszym) wykonaniu pętli porównuje się z zadaną przez użytkownika liczbą kroków, która została wcześniej wczytana do rejestru %r13.

W każdej iteracji pętli:

-Obliczana jest górna wartość ułamka:

#multiply POWER 2 times

fxch %st(2) #take x to the top of the stack

fmul %st(0), %st(1) #power\*=x

fmul %st(0), %st(1) #power\*=x

fxch %st(2) #take x back to place

-Obliczana jest dolna wartość ułamka:

#increase factorial

movq %r10, %rax #move factorial to rax

inc %r11 #highestSoFar++

mulq %r11 #rax\*=highestSoFar

inc %r11 #highestSoFar++

mulq %r11 #rax\*=highestSoFar

movq %rax, %r10 #factorial back to r10

-Obliczany jest ostateczny ułamek

fldz #load 0 to top of stack

fadd %st(2) #partialSum+=power

subq $16, %rsp #make space on stack

movq %r10, (%rsp) #load factorial on stack

fidiv (%rsp) #partialSum/=factorial

addq $16, %rsp #free the stack

-W zależności od parzystości licznika pętli, wartość ułamka jest dodawana lub odejmowana od ostatecznej sumy

movq %r12, %rax

CDQ #sign-extend rax to rdx

movq $2, %rbx

divq %rbx #i/2

cmp $0, %rdx #i%2==0

cmp:

jne odd

even:

faddp %st(0), %st(1) #sum+=partialSum and pop partialSum

jmp loopEnding

odd:

fsubrp %st(0), %st(1) #sum-=partialSum and pop partialSum

jmp loopEnding

Po zakończeniu wykonywania pętli wynik wypisywany jest na wyjście standardowe przy użyciu funkcji printf. Warto zauważyć że przekazywanie argumentów zmiennoprzecinkowych do funkcji zgodnie z ABI dokonywane jest porzez rejestry %xmm

movsd 8(%rsp), %xmm0 #move rads to xmm0 reg

movq $1, %rax #tell printf to accept 1 arg from xmm regs

movq $ResultString, %rdi #move RadString format address to rdi

call printf #print given input in rads